Readable semi-automatic formal proofs of Depth-First Search in graphs using Why3

Ran Chen and Jean-Jacques Lévy
State Key Laboratory for Computer Science,
Institute of Software, Chinese Academy of Sciences
& Inria

Abstract. Quite often formal proofs are not published in conferences or journal articles, because formal proofs are usually too long. A typical article states the ability of having implemented a formal proof, but the proof itself is often sketched in terms of a natural language. At best, some formal lemmas and definitions are stated. Can we do better ? We try here to publish the details of a formal proof of the white-paths theorem about depth-first search in graphs. We use Why3 as the proving platform, because Why3 uses first-order logic augmented with inductive definitions of predicates and because Why3 makes possible to delegate bits of proofs to on-the-shelf automatic provers at same time as Why3 provides interfaces with interactive proof checkers such that Coq, PVS or Isabelle. Algorithms on graphs are also a good testbed since graphs are combinatorial structures whose algebraic properties are not fully obvious. Depth-first search may look over-simple, but it is the first step of the construction of a library of readable formal proofs for more complex algorithms on graphs with more realistic data structures.

1 Introduction

Formal proofs of program correctness are a big challenge. They often comprise a large number of cases, which makes them intractable on paper. Fortunately proof-assistants and automatic provers can help. But the resulting proofs are usually long listings of elementary steps which are almost impossible to read by a normal human being. Even in the realm of mathematics where algebraic properties have been studied for a long time, formal libraries are hard to follow. Take for instance the impressive Mathematical Components library, the Standard Coq library or the Compcert certified compiler. It would be good to have readable quasi-formal proofs for algorithms or easy mathematics. Our paper aims to work in that direction.

We consider first-order logic with inductive definitions for predicates as such implemented in the Why3 platform [10]. First-order logic lacks of abstraction. For instance, we miss a calculus of relations which could be useful for graphs, but first-order logic is easy to understand and allows mechanical proofs. Higher-order logic allows conciseness, and with the help of the Coq proof-assistant, one can use elegant notations for operators. But we refer the reader to the proofs about finite graphs in the MathComp library [11], where one needs to

type *vertex*

constant *vertices* : *set vertex*

function *successors vertex* : *set vertex*

axiom *successors_vertices* : $\forall x.$ *mem x vertices* $\rightarrow$ *subset* (*successors x*) *vertices*

predicate *edge* (*x y* : *vertex*) = *mem x vertices* $\wedge$ *mem y* (*successors x*)

Fig. 1. Definitions and axioms for finite graphs

understand higher-order logic, the difference between propositions and truth-values, and small reflection. (And we did not mention the proofs with backward chaining)

In a previous work, we considered basic programs with lists and arrays [16], such as mergesort as implemented in Sedgewick's book about algorithms [21]. There is also the fantastic gallery of programs on the Why3 webpage [9], which enumerates many formal proofs of algorithms on arrays and algebraic structures. But here we consider graphs with its most basic program, i.e. depth-first search (dfs). We will treat three versions of dfs expressed in the ML Why language, which here would not be much different from a functional language, although ML Why also allows mutable data structures and imperative programming. The interested reader is also referred to our webpage (jeanjacqueslevy.net/why3) for more iterative versions of dfs, one of which corresponds exactly to the version in Sedgewick's book.

2 Representation of graphs

A graph is represented by a finite set of vertices and a successors function which gives for any vertex the finite set of vertices directly reachable from it (figure 1). The *edge* predicate states that there is an edge from its first argument to the second argument. The *mem* predicate expresses membership to a set. The finite set theory is presented in the Why3 standard library (located at URL why3.lri.fr/stdlib-0.86).

In this paper, we want to prove the white-paths theorem which often appears in books about algorithms. Therefore we need a theory of paths in graphs. We take paths as already defined in the graph theory of the Why3 standard library (see figure 2). Thus *path* x l z states that there is a path l from vertex x to vertex z in the graph. The path l is the list of intermediate vertices comprising the first vertex x but not the last one z, except when x and z are the same vertex. (We therefore use the polymorphic list library of Why3) We further define two predicates *reachable* and *access* useful for our first proof of dfs. The first predicate says there is a path between two vertices without precising the path, the second one says that its second argument is reachable from the first set of vertices.

$$
\begin{array}{l}
\textbf{inductive}\ \ path\ vertex\,(list\ \ vertex)\ \ vertex =\\
\ \ \mid\ \ Path\_empty : \forall x : vertex.\ \ path\ \ x\ \ Nil\ \ x\\
\ \ \mid\ \ Path\_cons :\\
\qquad \forall x\ y\ z : vertex,\ \ l : list\,vertex.\\
\qquad edge\ \ x\ y\ \ \rightarrow\ \ path\ \ y\ l\ z\ \ \ \rightarrow\ \ \ path\ \ x\ (Cons\ \ x\ l)\ z\\
\\
\textbf{predicate}\ \ reachable\,(x\ z :\ \ vertex) = \exists\, l.\ path\ \ x\ \ l\ \ z\\
\\
\texttt{lemma}\ \ reachable\_succ :\ \forall x\ x'.\ \ edge\ \ x\ x'\ \ \rightarrow\ \ reachable\ \ x\ x'\\
\\
\texttt{lemma}\ \ reachable\_trans :\ \forall x\ y\ z.\ \ reachable\ \ x\ y\ \ \rightarrow\ \ reachable\ \ y\ z\ \ \rightarrow\ \ reachable\ \ x\ z\\
\\
\textbf{predicate}\ \ access\,(r\ s :\ \ set\ \ vertex) =\\
\qquad \forall z.\ \ mem\ \ z\ s\ \ \rightarrow\ \exists x.\ \ mem\ \ x\ r\ \wedge\ reachable\ \ x\ z
\end{array}
$$

Fig. 2. Definitions and lemmas for Dfs with non-black-to-white assumption

3 Dfs with non-black-to-white assumption

Depth-first search is a naive recursive search on graphs which marks vertices to prevent from looping. It starts from a set r of roots and provides as result the set of vertices accessible from r. We use three types of marking: white, gray, black. White vertices are not yet visited, gray vertices started to be explored, black vertices are fully visited. As usual, dfs chooses randomly a vertex x in roots, turns that node to gray and explores the successors of x. Then dfs turns x to black and continues with remaining roots (see figure 3). In our program, the parameters of dfs are r, g, b standing for the sets of roots, gray vertices and black vertices. The functions *is_empty*, *choose*, *mem*, *union*, *add*, *remove* are standard functions of the set library in Why3. Notice that our program is not fully effective since working on sets rather than lists, but this presentation facilitates the correctness proof. (On our webpage there are proofs for more efficiently implementable versions of dfs) Here dfs contains two recursive calls, but the second call is tail recursive and we could have used a more iterative version.

In this version of dfs, we show the *no_black_to_white* invariant which says that an edge from a black vertex can only end into a non-white vertex, i.e. a black or gray node. We also prove as post-conditions of dfs that all vertices in the result are accessible from the initial sets of black and gray vertices, and further that non-gray roots belong to the result. The intermediate assertions are self-explainable ($b1$ is the resulting set of visited nodes after the recursive call on successors of x; and $b2$ is same set with x turned to black). These assertions and post-conditions are proved automatically by Alt-Ergo [3], Eprover [20], Spass [23] and Z3 [7]. The variant property states the termination of dfs (here with a lexicographic ordering on the pair made of the number of non-gray nodes and the number of roots)

```
predicate no_black_to_white (b g : set vertex) =
    ∀x x'. edge x x' → mem x b → mem x' (union b g)

let rec dfs r g b :
  variant{(cardinal vertices − cardinal g), cardinal r} =
  requires {subset r vertices}
  requires {subset g vertices}
  requires {no_black_to_white b g}
  ensures {subset b result}
  ensures {no_black_to_white result g}
  ensures {∀x. mem x r → ¬ mem x g → mem x result}
  ensures {access (union b r) result}

 if is_empty r then b
  else
  let x = choose r in
  let r' = remove x r in
  if mem x (union g b) then
    dfs r' g b
  else begin
    let b1 = dfs (successors x) (add x g) b in
    assert{access (add x b) b1};
    assert{access (union r b) b1};
    let b2 = add x b1 in
    assert{access (union r b) b2};
    dfs r' g (union b b2)
  end
```

Fig. 3. Dfs with non-black-to-white assumption (part I)

Now we want to prove that dfs results in the set of all nodes accessible from the roots when we start with empty sets of gray and black nodes. That is all vertices are white at the beginning of dfs (see figure 4). The post-conditions of *dfs_main* function are expressed in terms of white paths and node flipping, since we want to match this proof with following ones on other versions of dfs. The first post-condition means that accessible vertices from roots are in the result, the second post-condition is the other direction (all vertices in the result are accessible from roots). A white path is white with respect to a set v of visited vertices; the definition of *whitepath* uses the *L.mem* membership predicate on lists, which is distinct from the *mem* predicate on sets.

The two post-conditions of *dfs_main* are proved by Alt-Ergo, CVC3 [1] and Eprover, once lemma *no_black_to_white_nopath* is proved. This lemma states that there is no path from a black vertex to a white vertex, when the *no_black_to_white* condition holds. We then are forced to go through a gray node. This lemma needs an induction on the length of the path, which is difficult to get with SMT-solvers or even theorem provers. The keyword `‘‘induction’’` may be used to hint an induction on the preceding variable, but in our case it did not work and Coq has

```
predicate white_vertex (x : vertex) (v : set vertex) = ¬ (mem x v)

predicate nodeflip (x : vertex) (v1 v2 : set vertex) =
    white_vertex x v1 ∧ ¬ (white_vertex x v2)

predicate whitepath (x : vertex) (l : list vertex) (z : vertex) (v : set vertex) =
    path x l z ∧ (∀y. L.mem y l → white_vertex y v) ∧ white_vertex z v

predicate whiteaccess (r : set vertex) (z : vertex) (v : set vertex) =
    ∃ x l. mem x r ∧ whitepath x l z v

predicate nodeflip_whitepath (r v1 v2 : set vertex) =
    ∀z. nodeflip z v1 v2 → whiteaccess r z v1

predicate whitepath_nodeflip (r v1 v2 : set vertex) =
    ∀x l z. mem x r → whitepath x l z v1 → nodeflip z v1 v2

lemma no_black_to_white_nopath :
    ∀g b. no_black_to_white b g →
      ∀x l "induction" z. path x l z → mem x b → ¬ mem z (union b g) →
          ∃y. L.mem y l ∧ mem y g

let dfs_main r : =
  requires {subset r vertices}
  ensures {whitepath_nodeflip r empty result}
  ensures {nodeflip_whitepath r empty result}
  dfs r empty empty
```

Fig. 4. Dfs with non-black-to-white assumption (part II)

to be used (see our webpage). We use Coq with the Ssreflect package, although not yet fully compatible with the Why3 Coq driver. The Coq proof is then quite easy, since the argument is obvious; but it is Coq stylish.

4 Random search stepwise

The previous proof does not match the standard proofs for dfs which states a finer property known as the white-paths theorem [6]. Whatever is the initial marking of vertices, this theorem states that a vertex has its color flipped if and only if it is accessible from the roots by a white path. We first consider that property with one step of random search in a graph, as suggested to us by a note of Dowek quoting a proof by Muñoz [18, 8]. In the rest of this paper, we only use two colors as marks of vertices: white and black.

The random step search picks any white node x in the set of roots and replace it by its successors after marking x to black. We thus continue with the union of these successors of x and the set of roots minus x. If the picked vertex x is non-white, we remove it from the set of roots. When no more roots, we give the

```
let rec search1 r v
  variant{(cardinal vertices − cardinal v), cardinal r} =
  requires {subset r vertices}
  requires {subset v vertices}
  ensures {subset v result}
  ensures {nodeflip_whitepath r v result}
if is_empty r then
 v
else
 let x = choose r in
 let r' = remove x r in
 if mem x v then
   search1 r' v
 else
  let b = search1 (union r' (successors x)) (add x v) in
  (* ———————— nodeflip_whitepath ———————— *)
  assert{∀z. nodeflip z v b  →  z = x ∨ nodeflip z (add x v) b};
  (* case 1.1: nodeflip z v b ∧ z = x *)
   assert{whitepath x Nil x v};
  (* case 1.2: nodeflip z v b ∧ z ≠ x *)
   assert{∀z. nodeflip z (add x v) b → whiteaccess r' z (add x v)
                  ∨ whiteaccess (successors x) z (add x v) };
   assert{∀x' l z. whitepath x' l z (add x v) → whitepath x' l z v};
   assert{∀z x' l. edge x x' → whitepath x' l z v → whitepath x (Cons x l) z v};
   assert{∀z. nodeflip z (add x v) b → whiteaccess r z v};
  b
```

Fig. 5. Random search step (part I)

marked nodes as the result. This search step is compatible with various searching strategies (depth-first, breadth-first). It is also interesting to notice that this proof is generic of the further dfs proof that we will later consider. There is here an interesting analogy with the Lamport's way of proving quicksort with an iterative algorithm working on quicksort step (see Meyer's Lampsort [17]). We consider two independent proofs.

We first prove the *nodeflip_whitepath* post-condition (see figure 5) with Alt-Ergo and CVC3. This simple proof considers the case the flipped node z is the picked white vertex x or distinct from it. If z is x, then the empty *Nil* path is white at the beginning of the search step. If z is not x, then it is flipped by the recursive call with again two cases: the node is accessible from the successors of x or the rest of roots r'. In both case, we conclude to the existence of an initial white path by monotony on the last argument of *whitepath*.

The *whitepath_nodeflip* post-condition needs more work (see figure 6). Assume we have a white path from the picked white vertex x to another vertex z. The case $z = x$ is easily solved by the first post-condition proving that x belongs to the result b. When z and x are distinct, we rely on the important lemma *whitepath_whitepath_fst_not_twice* (see figure 9). We know then that there is a

```
let rec search1 r v
  variant{(cardinal vertices − cardinal v), cardinal r} =
  requires {subset r vertices}
  requires {subset v vertices}
  ensures {subset v result}
  ensures {whitepath_nodeflip r v result}
if is_empty r then
 v
else
 let x = choose r in
 let r' = remove x r in
 if mem x v then
   search1 r' v
 else
  let b = search1 (union r' (successors x)) (add x v) in
  (* ————————— whitepath_nodeflip ———————— *)
  (* case 1: whitepath x l z v ∧ x = z *)
   assert {mem x b};

  (* case 2: whitepath x l z ∧ x ≠ z *)
   (* using lemma whitepath_whitepath_fst_not_twice *)
   assert {∀l z. whitepath x l z v → x ≠ z
      → ∃ x' l'. edge x x' ∧ whitepath x' l' z (add x v) };
   assert {∀l z. whitepath x l z v → x ≠ z → nodeflip z (add x v) b};
  assert {∀l z. whitepath x l z v → nodeflip z v b};

  (* case 3: whiteaccess r' z v *)
   (* case 3.1: whitepath r' l z ∧ (L.mem x l ∨ x = z) *)
    assert {∀y l z. whitepath y l z v → (L.mem x l ∨ x = z)
      → ∃l'. whitepath x l' z v};
    (* goto cases 1.2 *)
   (* case 3.2: whitepath r' l z ∧ ¬(L.mem x l ∨ x = z) *)
    assert {∀y l z. mem y r' → whitepath y l z v → ¬(L.mem x l ∨ x = z)
      → whitepath y l z (add x v)};
  b
```

Fig. 6. Random search step (part II)

white path from x to z not containing x. Therefore by the recursive call we knows that z has been flipped. Now if the white path was starting not from x but from another vertex in the set of remaining roots r'. If that path contains x, it is no longer white when starting the recursive call. But that means that there was a white path from x to z and we go back to the case of white paths issued from x. If that path from r' does not contains x, it is still white at the recursive call and inductively the node z at end of the path is flipped. That proof is automatic with Alt-Ergo, CVC3 and Eprover. The lemmas can also be

```
let rec dfs r v
  variant{(cardinal vertices − cardinal v), cardinal r} =
  requires {subset r vertices}
  requires {subset v vertices}
  ensures {subset v result}
  ensures {subset result vertices}
  ensures {nodeflip_whitepath r v result}
  ensures {whitepath_nodeflip r v result}
if is_empty r then
 v
else
 let x = choose r in
 let r' = remove x r in
 if mem x v then
   dfs r' v
 else
 let b = dfs (successors x) (add x v) in
 let b' = dfs r' b in
 (* ———— nodeflip_whitepath ———————————————— *)
  assert{∀z. nodeflip z v b' → nodeflip z v b ∨ nodeflip z b b'};
  (* ———— case 1 ————— *)
  assert{∀z. nodeflip z v b → z = x ∨ nodeflip z (add x v) b};
  (* case 1.1: nodeflip z v b ∧ z = x *)
   assert{whitepath x Nil x v};
  (* case 1.2: nodeflip z v b ∧ z ≠ x *)
   assert{∀z. nodeflip z (add x v) b → whiteaccess (successors x) z (add x v)
   assert{∀x' l z. whitepath x' l z (add x v) → whitepath x' l z v};
   assert{∀z x' l. edge x x' → whitepath x' l z v → whitepath x (Cons x l) z v};
   assert{∀z. nodeflip z (add x v) b → whiteaccess r z v};
  (* ———— case 2 ————— *)
   assert{∀z. nodeflip z b b' → whiteaccess r' z b};
   assert{∀z x' l. whitepath x' l z b → whitepath x' l z v};
...
```

Fig. 7. Depth-First search (part I)

proved automatically, but we will discuss about the proof of these lemmas in later section.

Notice that we can prove exactly in the same way the iterative dfs algorithm with a stack, or breadth-first search with a queue (see our webpage) One just needs to add two assertions about the state of the stack or queue. It is frustrating that the recursive version of dfs is a bit longer to be proved!

5 Depth-first search

The program is now making the standard recursive call on the successors of the picked white vertex x and another tail-recursive call on the remaining set r' of

```
...
 (* ———— whitepath_nodeflip ———————————————— *)
 (* case 1: x = z *)
  assert{mem x b};
 (* case 2: ∃l. whitepath x l z v ∧ x ≠ z *)
 (* using lemma whitepath_whitepath_fst_not_twice *)
  assert{∀l z. whitepath x l z v → x ≠ z →
          whiteaccess (successors x) z (add x v) };
  assert{∀l z. whitepath x l z v → x ≠ z → nodeflip z (add x v) b};
  assert{∀l z. whitepath x l z v → nodeflip z v b};
 (* case 3: whiteaccess r' z v *)
  (* case 3.1: ¬ whiteaccess r' z b *)
   assert{∀x' l z. whitepath x' l z v → ¬ whitepath x' l z b →
          ∃y.(L.mem y l ∨ y = z) ∧ nodeflip y v b};
   assert{∀x' l z. whitepath x' l z v → ¬ whitepath x' l z b →
    ∃y.(L.mem y l ∨ y = z) ∧ (y = x ∨ whiteaccess (successors x) y (add x v))};
   assert{∀y. whiteaccess (successors x) y (add x v) →
          ∃l'. whitepath x l' y v};
   assert{∀x' l z. whitepath x' l z v → ¬ whitepath x' l z b →
          ∃y l'.(L.mem y l ∨ y = z) ∧ whitepath x l' y v};
   assert{∀x' l z. whitepath x' l z v → ¬ whitepath x' l z b →
          ∃l'. whitepath x l' z v};
   (* goto cases 1-2 *)
   assert{∀x' l z. whitepath x' l z v → ¬ whitepath x' l z b → nodeflip z v b'};
  (* case 3.2: whiteaccess r' z b*)
   assert{∀x' l z. mem x' r' → whitepath x' l z b → nodeflip z v b'};
 b'
```

Fig. 8. Depth-First search (part II)

roots with the already visited vertices augmented by the set of nodes visited by the call on the successors of x. As the already visited nodes are part of the result, one has just to consider the result b of this recursive call. We again split the proof in two proofs with respect to the direction of post-conditions.

The proof of *nodeflip_whitepath* is quite similar to the one of the random search step (see figure 7). There is an extra case when the node is flipped during the tail-recursive call, which is just proved inductively by the post-condition of that call.

The proof of *whitepath_nodeflip* is more subtle. The difference comes from the larger set of nodes which are flipped by the recursive call. In random search step, we knew that only the picked node x was flipped before the tail-recursive call. Here the whole set produced by the dfs call on successors of x is flipped. So we start as in the random search step by considering a white path from x to any vertex z. If the path starts from x, the proof is similar to the one of random search step. But if the path starts from a vertex in r', the proof is a bit more complex. If the path keeps white after the recursive call on successors of x (case 3.2 in the proof in figure 8), then the post-condition of the tail-recursive call states that z is flipped. If the path is no longer white after the recursive call on successors of x (case 3.1 in the proof), then there is a node y which flipped on that path after the first recursive call. That node y is either x or is flipped with respect to *add* x v. Then by using the *nodeflip_whitepath*, we know that y is connected by a white path from x or one of its successors. In both cases, there is a white path starting from x and leading to y. Therefore there is a path from x to z by using lemma *whitepath_Y*. We thus notice that these two proofs of the post-conditions of dfs are not independent.

We now discuss definitions and lemmas needed by these dfs proofs. We first use the *whitepath_fst_not_twice* predicate. Notice we need not use simple paths, i.e. paths with no vertex repetitions. Eliminating repetition of the initial node is sufficient. (Muñoz's note is using simple paths) We can also prove automatically the lemma following lemma *path_suffix_fst_not_twice* with the useful `''induction''` keyword, hinting induction to be done on previous variable, which Eprover can follow. All lemmas are proved automatically (see the stats in figure 10). Lemma *whitepath_Y* looks strange, but is quite useful when a white path has an intermediate node which is flipped.

```
predicate path_fst_not_twice (x : vertex) (l : list vertex) (z : vertex) =
    path x l z ∧
    match l with
    | Nil  →  true
    | Cons _ l'  →  x ≠ z ∧ ¬ L.mem x l'
    end

lemma path_suffix_fst_not_twice :
  ∀x z l "induction". path x l z → ∃l1 l2. l = l1 ++ l2 ∧ path_fst_not_twice x l2 z

lemma path_path_fst_not_twice :
  ∀x z l. path x l z →
     ∃l'. path_fst_not_twice x l' z ∧ subset (E. elements l') (E. elements l)

predicate whitepath_fst_not_twice (x : vertex) (l : list vertex) (z : vertex)
  (v : set vertex) = whitepath x l z v ∧ path_fst_not_twice x l z

lemma whitepath_whitepath_fst_not_twice:
  ∀x z l v. whitepath x l z v → ∃ l'. whitepath_fst_not_twice x l' z v
...
lemma whitepath_trans:
  ∀x l1 y l2 z v. whitepath x l1 y v → whitepath y l2 z v →
         whitepath x (l1 ++ l2) z v

lemma whitepath_Y:
  ∀x l z y x' l' v. whitepath x l z v → (L.mem y l ∨ y = z) →
         whitepath x' l' y v → ∃ l0. whitepath x' l0 z v
```

Fig. 9. Definitions and lemmas for dfs

6 Conclusion

We hope to have met our goal of producing readable formal proofs (checked by computer) for depth-first search in graphs. There are surely other formal proofs such as the ones by Neuman (180 lines of Isabelle) [14] or Pottier (in Coq) [19]. Depth-first search can also be implemented with more concrete data structures. We proved a version with arrays and lists as in algorithms textbooks. We can also easily design one with lists for sets as the sequences in the Mathcomp library. One longer term objective is to get readable proofs for other algorithms on graphs. We do have proofs of test for acyclicity, strongly connected components with various techniques and minimum spanning tree. Readable versions of these programs proofs have to be soon inserted on our webpage.

The engineering of readable long proofs is less clear. Why3 is a fabulous system for interfacing many provers and interactive proof systems. But making these proofs is often unstable. An interactive proof assistant where all elementary steps are explicit (as long as implicit names are not permitted such as in MathComp) is more robust to modifications. Again a better system of notations (as in the Coq system) would help for the readability of proofs. Finally we did not use ghost variables in the proofs presented in this article, but these are quite useful in many proofs of algorithms. Ghost propositions would also help to maintain a set of valid hypotheses in the proof environments.

7 Acknowledgements

We thank Jean-Christophe Filliâtre and the Why3 team for their precious advices. We are grateful to Gilles Dowek for pointing us his note with Cesar Muñoz. J.-J. Lévy deeply thanks Huimin Lin and ISCAS for hosting this research.

References

1. C. Barrett and C. Tinelli. CVC4, the smt solver. New-York University - University of Iowa, 2011. http://cvc4.cs.nyu.edu.
2. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Springer, 2004.
3. F. Bobot, S. Conchon, E. Contejean, M. Iguernelala, S. Lescuyer, and A. Mebsout. The alt-ergo automated theorem prover, 2008. http://alt-ergo.lri.fr/.
4. F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011.
5. Coq Development Team. The coq 8.5 standard library. Technical report, Inria, 2015. http://coq.inria.fr/distrib/current/stdlib.
6. T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms.* McGraw-Hill Higher Education, 2nd edition, 2001.
7. L. de Moura and N. Björner. Z3, an efficient smt solver. Microsoft Research, 2008. http://z3.codeplex.com.

8. G. Dowek et al. *Informatique et sciences du numérique- Spécialité ISN en terminale S*. Eyrolles, 2012.
9. J.-C. Filliâtre et al. The why3 gallery of verified programs. Technical report, CNRS, Inria, U. Paris-Sud, 2015. http://toccata.lri.fr/gallery/why3.en.html.
10. J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In M. Felleisen and P. Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, Mar. 2013.
11. G. Gonthier et al. Finite graphs in mathematical components, 2012. Available at http://ssr.msr-inria.inria.fr/~jenkins/current/Ssreflect.fingraph.html, The full library is available at http://www.msr-inria.fr/projects/mathematical-components-2/.
12. G. Gonthier, A. Mahboubi, L. Rideau, E. Tassi, and L. Théry. A modular formalisation of finite group theory. In *Theorem Proving in Higher-Order Logics (TPHOLS'07)*, volume 4732 of *Lecture Notes in Computer Science*, pages 86–101, 2007.
13. G. Gonthier, A. Mahboubi, and E. Tassi. A Small Scale Reflection Extension for the Coq system. Rapport de recherche RR-6455, INRIA, 2008.
14. P. Lammich and R. Neumann. A framework for verifying depth-first search algorithms. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, CPP '15, pages 137–146, New York, NY, USA, 2015. ACM.
15. X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 2009. http://compcert.inria.fr.
16. J.-J. Lévy. *Essays for the Luca Cardelli Fest*, chapter Simple proofs of simple programs in Why3. Microsoft Research Cambridge, MSR-TR-2014-104, 2014.
17. B. Meyer. Lampsort. Technical report, ETHZ, 2014. http://bertrandmeyer.com/2014/12/07/lampsort.
18. C. Muñoz, V. Carreño, and G. Dowek. Formal analysis of the operational concept for the Small Aircraft Transportation System. In *Rigorous Engineering of Fault-Tolerant Systems*, volume 4157 of *Lecture Notes in Computer Science*, pages 306–325, 2006.
19. F. Pottier. Depth-first search and strong connectivity in Coq. In *Journées Francophones des Langages Applicatifs (JFLA 2015)*, Jan. 2015.
20. S. Schulz. System Description: E 1.8. In K. McMillan, A. Middeldorp, and A. Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*. Springer, 2013.
21. R. Sedgewick and K. Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011.
22. A. Tafat and C. Marché. Binary heaps formally verified in Why3. Research Report 7780, INRIA, Oct. 2011. http://hal.inria.fr/inria-00636083/en/.
23. C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. Spass version 3.5. In *22nd International Conference on Automated Deduction, CADE 2009*, number 5663 in LNCS, pages pp. 140–145, 2009.

Proof obligations		Alt-Ergo (0.95.2)	CVC3 (2.4.1)	Eprover (1.8-001)	Eprover (1.9)	Spass (3.5)	Z3 (4.3.2)	Z3 (4.4.0)
path_suffix_fst_not_twice		(15s)	(15s)	11.47	19.09	(15s)	(15s)	
path_path_fst_not_twice		(15s)	(15s)				0.05	0.02
whitepath_decomposition		(15s)	0.65		(15s)			
whitepath_mem_decomposition_r		(45s)	(45s)		(45s)	4.03		No run
whitepath_whitepath_fst_not_twice		(15s)	(65s)	(65s)			0.02	(65s)
	1.							0.02
path_cons_inv		0.04	0.09		(15s)			
whitepath_cons_inv		(15s)	(15s)		0.88			
whitepath_cons_fst_not_twice_inv		(15s)	(15s)		(15s)	(15s)		(15s)
	1.				0.07			
whitepath_fst_not_twice_inv		(15s)	(15s)	0.04	0.18	(15s)	(15s)	
whitepath_trans		0.06				(15s)		
whitepath_Y			(45s)			4.30		
VC for dfs	1. postcond	0.03						
	2. postcond	0.03						
	3. postcond	0.03						
	4. postcond	0.03						
	5. variant	0.05						
	6. precond	0.03						
	7. precond	0.02						
	8. postcond	0.03						
	9. postcond	0.03						
	10. postcond		0.97					
	11. postcond	0.04						
	12. variant	0.05						
	13. precond	0.04						
	14. precond	0.04						
	15. variant	0.08						
	16. precond	0.03						
	17. precond	0.03						
	18. assertion	0.03	0.06			0.07		0.02
	19. assertion	0.05						
	20. assertion	0.03						
	21. assertion		0.05					
	22. assertion	14.87						
	23. assertion	0.17				(35s)		
	24. assertion							0.04
	25. assertion		0.10					
	26. assertion		0.08					
	27. assertion	0.05						
	28. assertion		1.15					
	29. assertion	11.26	0.80					
	30. assertion	0.31	No run					No run
	31. assertion					0.42		
	32. assertion		0.90					
	33. assertion		6.32					
	34. assertion		1.31					
	35. assertion	(15s)	(15s)		(45s)	(45s)		0.05
	36. assertion		No run					0.08
	37. assertion		0.25					
	38. postcond		4.85			No run		No run
	39. postcond	0.03	0.09			No run		No run
	40. postcond		4.61					No run
	41. postcond		3.55					
VC for dfs_main	1. precond	0.02						
	2. precond	0.03						

Fig. 10. Stats of the Depth-first search proof on 2.93 GHz Intel Core 2 Duo